Toward accurate clock drift modeling in Wireless Sensor Networks simulation

David Hauweele[a,*], Bruno Quoitin[a]

[a] *University of Mons (UMONS), Mons, Belgium*

Abstract

The protocols used in Wireless Sensor Networks are subject to very strict temporal synchronization constraints. Radio Duty Cycle (RDC) protocols in particular are characterized by their very high synchronization accuracy requirements. In these protocols, synchronization errors are primarily caused by the natural clock drift observed in real nodes. The impact of the clock drift on the desynchronization issues can be investigated by the use of low-level node simulators.

In this paper, we show the limitation of the COOJA simulator to evaluate the impact of the clock drift. We show that its current mote execution model does not faithfully reproduce the requested clock drift. We identify the root cause of these inaccuracies and present a new algorithm that is able to precisely reproduce any requested clock drifts in simulation. On the basis of our understanding, we build a mathematical model of the clock drift error allowing previous authors to estimate the error caused by clock drift inaccuracies in their studies and its impact on their results.

To demonstrate the new algorithm, we consider a well-documented RDC protocol issue where the clock drift would cause periodic communication blackouts. This phenomenon was observed on real nodes, but was previously impossible to reproduce by simulation. Our new algorithm not only allows to reproduce the blackouts but also provides additional insights that help to confirm the initial analysis of the phenomenon. Finally, we use our algorithm to implement dynamic clock drift models to simulate the behavior of a clock drift evolving through time. We illustrate this using our linear model to push a RDC protocol to its limits.

Keywords: clock drift, wireless sensor network, simulation

1. Introduction

The nodes used in Wireless Sensor Networks (WSNs) have limited power resources. They must however stay autonomous for extended periods of time. In order to achieve their low power requirements, the Medium Access Control (MAC) protocols used often rely on time synchronization mechanisms to allow the radio to

[*] Corresponding author

Email addresses: david.hauweele@umons.ac.be (David Hauweele), bruno.quoitin@umons.ac.be (Bruno Quoitin) alternate between active and sleep periods. This process, known as Radio Duty Cycling (RDC), greatly reduces the power consumption of the nodes, allowing the radio to remain off most of the time.

A wide variety of such protocols have been proposed in the literature using variants of the Radio Duty Cycling mechanism to save energy [YH04]. The main difficulty is to ensure that the motes wake-up at the approriate time to ensure successful communication while still limiting idle listening, overhearing, and collisions and doing so with minimal overhead. The proposed solutions can generally be divided into two categories. Synchronous solutions such as TSCH [iee15] organize the wake-up time of the motes so that they are ready to transmit and receive messages on a known schedule. On the other hand, asynchronous solutions such as ContikiMAC [Dun11] do not organize the wake-up time and let all motes run on their own schedule except for protocol constraints. In both cases the precision of those protocols is dependent on the precision of the clock used on the mote. Hence better clock synchronization usually translates to lower power consumption and reduced packet loss.

The impact of the clock drift on those protocols can be studied with real nodes [EDF$^+$16]. However this type of experiment is difficult for several reasons. First injecting a precise frequency to emulate the clock drift per node requires dedicated hardware. Second if the experiment highlights a statistical behavior, the actual clock drift per node must be precisely measured. Even so the real clock drift can still evolve by a great margin with time and temperature. Lastly due to hardware cost and the difficulty of setting up, these experiments are generally limited to a handful of nodes and in duration.

The use of discrete event simulation allows to overcome much of this complexity along with complete control of the environment parameters within the limits of the models provided in the simulator. In this paper, we evaluate the limited support for clock drift modeling in one of the prominent simulators, COOJA and its companion node emulator, MSPSim. We first present inaccuracies observed in simulation and quantify to what extent this affects the results of experiments. We then propose and implement an alternative model that provides more accurate clock drift modeling. To demonstrate its usefulness, we use it to reproduce a well-documented RDC protocol synchronization issue [BSTS17].

Our paper is organized as follows. Section 2 presents the clock drift problem and the typical oscillator architecture of a mote. Related work is discussed in Section 3. Then we describe in Section 4 the execution model used in COOJA to emulate motes, as well as the existing clock drift model. In Section 5 we present an experiment to validate the simulation of the clock drift in COOJA and the inaccuracies we detected. We follow, in the same section, with a description of the origin of the problem along with a mathematical model that can predict the observed inaccuracies. We present and evaluate, in Section 6, a new implementation for accurate modeling of the clock drift in WSN simulation. Using this new model, we reproduce a synchronization issue that affects ContikiMAC in Section 7. In Section 8, we implement dynamic clock drift models to simulate the behavior of a clock drift evolving through time which we illustrate using our linear model to push the ContikiMAC protocol to its limit. Finally, in Section 9, we draw concluding remarks and suggest further work.

2. Clock sources inaccuracies

The nodes used in Wireless Sensor Networks rely extensively on clock sources to regulate the passing of events. These clock sources are typically crystal-based oscillators. Ideally a noise-free non-drifting oscillator would generate a pure periodic signal at its output. However oscillators are influenced to some extent by their environment and manufacturing process tolerances. As a result, ambient temperature, supply voltages, magnetic fields or physical vibrations, to mention a few, can change the observed frequency.

2.1. Clock drift

The problem of clock source stability and its characterization has been regularly studied since the early-1960's [CS66, B$^+$71, A$^+$87]. To account for clock errors caused by manufacturing tolerances and the environment, a simple model for an oscillator is

$$V(t) = (V_0 + \varepsilon_V(t)) \sin[2\pi\nu_0 t + \phi(t)] \tag{1}$$

where V_0 is the signal amplitude, ν_0 its nominal frequency and $\phi(t)$ denotes the fractional phase or phase noise and $\varepsilon_V(t)$ the amplitude noise, but is negligible in most situations. From this we define the instantaneous frequency

$$\nu(t) = \frac{1}{2\pi}\frac{d}{dt}[2\pi\nu_0 t + \phi(t)] = \nu_0 + \frac{1}{2\pi}\frac{d\phi(t)}{dt} \tag{2}$$

It is useful to define the dimensionless ratio of frequency fluctuation, that is the normalized deviation of $\nu(t)$ from its nominal value

$$y(t) = \frac{\nu(t) - \nu_0}{\nu_0} = \frac{\Delta\nu(t)}{\nu_0} = \frac{1}{2\pi\nu_0}\frac{d\phi(t)}{dt} \tag{3}$$

It follows the time deviation of the oscillator

$$x(t) = \int_0^{t'} y(t')dt' = \frac{\phi(t)}{2\pi\nu_0} \tag{4}$$

This deviation is generally considered to be a combination of first a systematic, deterministic trend whose parameters can be predicted and second a random noise, characterized as a nondeterministic function of time. In [A$^+$87], the authors model these two contributions as

$$x(t) = \underbrace{x_0 + y_0 t + 1/2\ Dt^2}_{\text{systematic trend}} + \underbrace{\varepsilon_x(t)}_{\text{random noise}} \tag{5}$$

where $x(t)$ is the time deviation of the clock, x_0 the time offset, y_0 the frequency offset and D the frequency drift. In the context of this paper we are interested in the y_0 frequency offset parameter, a dimensionless ratio of deviation from the nominal frequency of the oscillator which we denote here as the *clock drift*.

2.2. Microcontroller clock system

Microcontrollers at the heart of Wireless Sensor Network nodes are often equipped with a complex clock system composed of one or more oscillators that can be configured as multiple clock sources. Those can be subdivided into three types as presented on Figure 1. First the *main system clock* (MCLK) which orders the execution of instructions on the CPU. Similarly the *subsystem clock* (SMCLK) is used for serial communication, peripherals and high-frequency timers. Finally the *auxiliary clock* (ACLK) is used for low-frequency events, timers and synchronization purposes. Each of these clocks can source their input from the different oscillators present on the system or eventually from an external input.

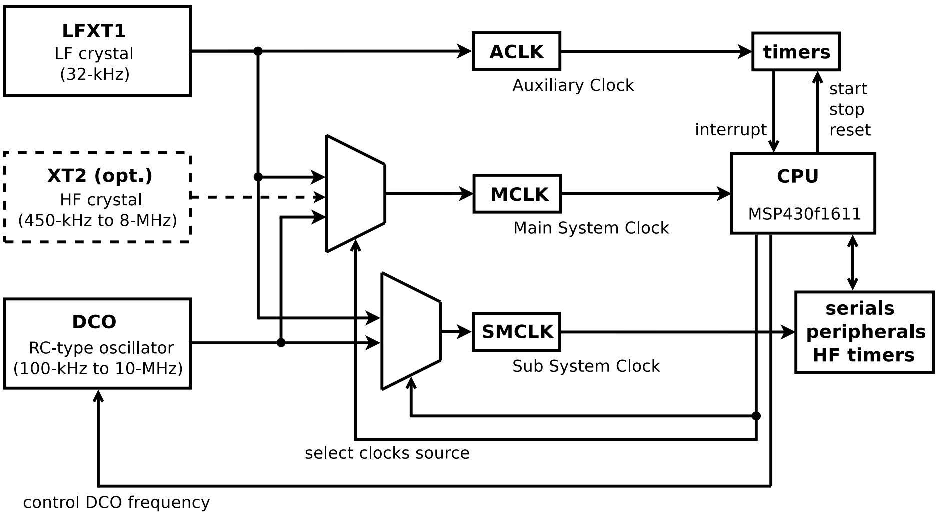


Figure 1: Clock hierarchy in the MSP430f1611 MCU.

In our experiments we use the default clock source attribution specified for the MSP430f1611 MCU used in the TMote Sky mote type. That is the auxiliary clock is sourced from a low-frequency 32-kHz crystal and the main and subsystem clocks are sourced from a high-frequency Digitally-Controlled Oscillator (DCO) synced at 4-MHz. These can be disabled along with associated circuitry and peripheral to achieve low power consumption in a sleep operating mode also known as Low Power Mode (LPM). For instance in LPM3, only the low-power and low-frequency crystal oscillator is enabled to allow internally programmed events such as timers to wake up the CPU. The mote firmware also uses the more precise low-frequency crystal oscillator to calibrate the frequency of the DCO. This process is regularly reiterated to account for the DCO higher variability. As a result the master clock also reflects eventual errors observed on the crystal oscillator.

3. Related work

Numerous simulators dedicated to the study of Wireless Sensor Networks are available [MPRS16]. Among those, instruction-level simulators combine a mote emulator which allows executing the firmware as it would be on a real device and a network wide simulator which handles the coordination of events such as radio transmission between multiple motes. However, to our knowledge only a few instruction-level simulators provide a mechanism to configure the clock drift.

In this paper, we focus on the COOJA [ODE+06] simulator along with its companion mote emulator, MSPSim. COOJA is one of the most used simulators as is attested by a recent survey [KKCP17] which shows that 63 % of the papers on RPL routing protocol studies relied on COOJA to run their simulations. Although TI MSP430 based motes are quite old compared to modern solutions available for use in WSN, they are still frequently used along with COOJA for experiments and prototyping purposes [KPK+18, AHWR18, RPGJ18, IVML18, PPS+18]. COOJA specifies a clock deviation parameter which can be individually configured for each mote in the simulation. As an example, COOJA was used in [PMF+16] to induce a clock error in the nodes and assess the performance of a proposed optimisation of TSCH. High-level and low-level simulators are compared using a statistical approach in [GHB+19] to characterize the limitations of the high-level model. In particular they require that the low-level simulator be able to model the clock drift.

Renode [Ltd18], a more recent mote emulator, supports more modern platforms but is not widely used at this time. It does not seem to provide support for clock deviation. However, its simulation model is similar to that of COOJA/MSPSim, meaning that it could be applied the same approach as in this paper. WSim and WSNet from the Worldsens environment [FCF07] allow to configure the drift individually on each clock of the motes by using a nanosecond simulation step. However this simulator is seldom used and is unmaintained since 2011. The OpenWSN [WVK+12] framework provides support for mote-specific clock drift in its dedicated simulator, OpenSim. It is unclear however how the drift is modeled in OpenSim. Omnet++ and the Castalia extension support modeling the clock drift [FMT10]. OpenSim and Omnet++ are not instruction-level simulators. That is, they are unable to run the firmware of motes unmodified. The 6TiSCH simulator [MDV+18] implements a careful abstraction of the 6TiSCH network stack and provides a model of clock drift and its impact on TSCH. However this is not an instruction-level simulator, and it only simulates the behavior of the network stack at a high level.

The following results were originally presented in MSWiM 2019 [HQ19]. In this version of our work, we add a more thorough explanation of the clock drift problem and discuss the typical clock architecture of a mote. We present the construction of a mathematical model to predict the clock drift inaccuracies. We study the evolution of the model parameter α for a range of typical firmwares in Contiki OS and show the impact of this parameter on the clock drift inaccuracies. We provide additional explanations and refined results for the ContikiMAC blackouts experiment. Finally, using our new algorithm, we implement dynamic clock drift models to simulate the behavior of a clock drift evolving through time. To illustrate this, we use our linear model to test the limit of ContikiMAC in the case of increasing clock deviation.

4. COOJA/MSPSim Execution model

In this section we present the model for concurrent execution of nodes in the COOJA simulator and MSPSim MSP430 mote emulator. Later, in Section 4.2, we present the current implementation of the deviation parameter, the model component responsible for the simulation of the clock drift.

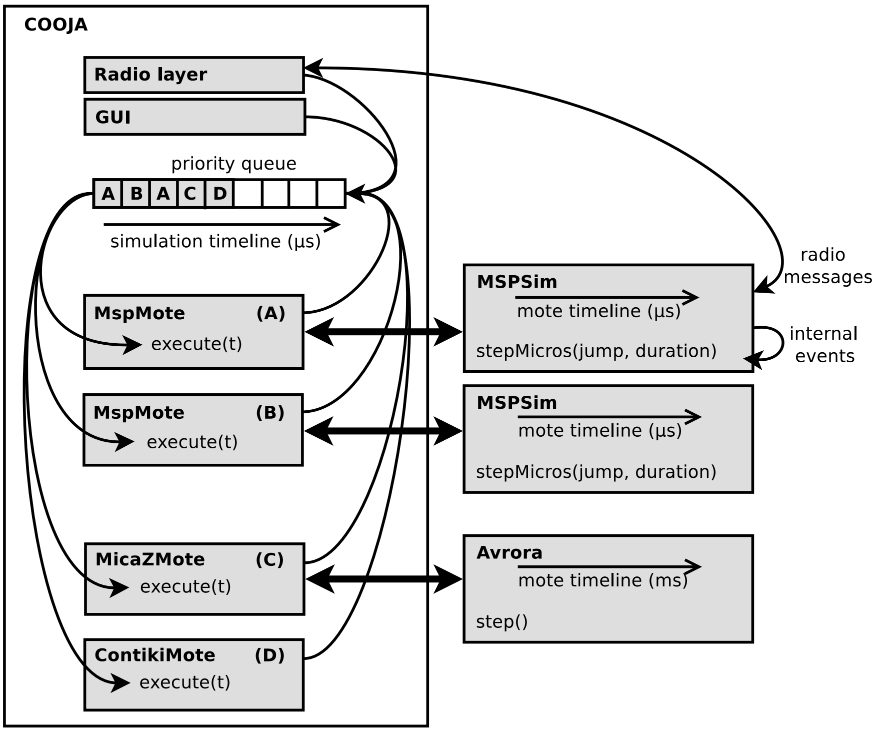


Figure 2: The COOJA/MSPSim simulator.

Figure 2 presents a high-level overview of the interactions between COOJA and MSPSim. Two timelines are involved during the simulation, a *simulation timeline* for the set of all motes and a *mote timeline* for each mote emulator. Both timelines have a microsecond precision and are discrete for performance reasons.

In COOJA, a priority queue schedules the executions of the motes along the simulator-specific timeline. Further execution on this queue are scheduled either by the motes own internal events, for instance timers, or scheduled by external events such as physical interactions from the GUI, peripherals and exchanged radio messages.

Each MSP mote in the simulation is coupled with an instance of the MSPSim mote emulator with its own distinct timeline. Normally both the mote and simulation timelines stay synchronized throughout the simulation and on each execution step the simulator advances the mote timeline proportionally to simulation time.

The same coupling also exists for the Avrora AVR motes emulator albeit with a different timeline precision. Contiki motes on their part run native versions of the Contiki OS. Hence their internal timelines are related to the host running the simulator. This distinction between simulation and emulation facilitates the integration of third-party emulators. It also enables the implementation of clock drift, described in Section 4.2, when not directly supported by the emulator as it is the case with MSPSim and Avrora.

In COOJA, the configuration of the clock deviation parameter induces a similar clock drift on the signal at the output of all source oscillators, that is LFXT1, XT2 and DCO. Thus the deviation parameter similarly impacts the timing of the instruction evaluations by the CPU, the duration of the timers and the behavior of peripherals.

4.1. Execution without deviation

We first detail the execution model of COOJA and the mote emulator MSPSim when no clock deviation is requested. Figure 3 illustrates the COOJA execution model for two simulation steps. The top timeline belongs to COOJA while the bottom timelines belong to a single emulated mote. The *cycles* timeline represents the internal mote time in term of CPU cycles.

At time t_0, the mote is scheduled for execution in the simulator. This triggers a period Δ, fixed in COOJA to 1 µs, for the emulator to evaluate instructions. To this end, MSPSim fixes itself a deadline Δ µs in the future during which it executes instructions and registers the elapsed cycles until the deadline is exceeded. At the end of this period, MSPSim reports the resulting possible advance on its own timeline to the simulator which accordingly schedules the next mote execution.

When MSPSim cycles count reaches the deadline c_1, it does not change its mote timeline yet. Hence the internal mote timeline is still at t_0. Instead it returns and waits for COOJA to provide it on the next execution step with the jump value necessary to keep track of the time advance. So at the next execution

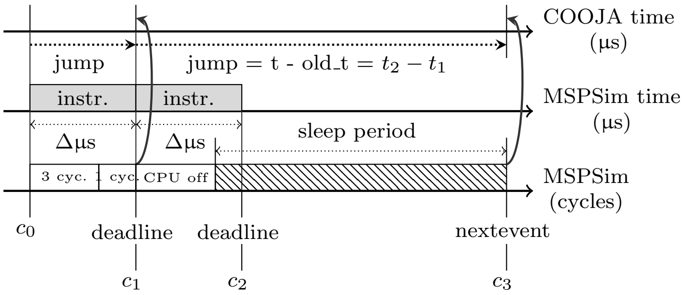


Figure 3: Two execution steps in COOJA and MSPSim.

step t_1, COOJA reports to MSPSim that Δ µs has elapsed since its last execution. Accordingly MSPSim advances its mote timeline of Δ µs allowing both simulation and mote timelines to resynchronize.

At t_1 simulation time, COOJA again executes the mote and the same instruction evaluation process is reiterated. Now however one instruction switches the CPU off, triggering a long sleep period. From programmed timers, the emulator knows the exact cycles count c_3 at which the next internal event should be triggered and reports this duration to the simulator. Hence the simulator only schedules the next mote execution until t_2, taking the reported sleep period into account. This allows both simulator and emulator to avoid execution during sleep periods resulting in a substantial speedup.

This also explains why the responsability for the mote timeline advance is left to the simulator. Should an external interrupt occur during the sleep period, the simulator must be able to prematurely wake-up the mote, providing it with a jump value shorter than the initially intended sleep period.

Listing 1 and 2 present a simplified version of the execution model as it is currently implemented in COOJA and MSPSim. The grayed parts are only specific to the deviation parameter presented in the next section and can be ignored for this section.

```
 1 MOTE_PARAMETER(deviation)
 2 skipped = 0                                        deviation
 3 total   = 0
 4
 5 old_t = 0
 6 Δ = 1 # µs
 7
 8 def execute(t):
 9   jump = t - old_t
10
11   if (1 - deviation) * total > skipped:
12     skipped += 1
13     jump    -= 1                                   deviation
14     scheduleNextExecution(t + 1)
15   total += 1
16
17   step_duration  = MSPSim.stepMicros(jump, Δ)
18   step_duration += Δ
19
20   scheduleNextExecution(t + step_duration)
21
22   old_t = t
```

Listing 1: Execution of a mote in COOJA.

The execution model for COOJA presented in Listing 1 is very simple. The simulator retains the last mote execution in the global variable `old_t`. Line 9 uses this variable to calculate the duration since the last execution, which we call the *jump* value. Line 17 triggers an execution step of Δ µs and provides MSPSim with the jump value previously calculated. MSPSim then reports the resulting possible advance on its mote timeline which the simulator uses to schedule the next mote execution on line 20.

```
1  cycles = 0
2  micros = 0
3
4  def stepMicros(jump, duration):
5    micros   += jump
6    deadline  = ((micros + duration) * mcuFrqMHz)
7
8    while (cycles < deadline):
9      if cpuoff:
10       cycles  = MIN(deadline, nextevent)
11     else:
12       cycles += emulateOP()
13
14   if cpuoff:
15     sleep_cycles = nextevent - cycles
16     return sleep_cycles / mcuFrqMHz
17   else:
18     return 0
```

Listing 2: Execution of a mote in MSPSim.

Listing 2 presents the emulator side of the execution model. Line 5 uses the jump value provided by COOJA to resynchronize the mote timeline. Line 6 fixes a deadline for a Δ µs instruction evaluation period. Those instructions are evaluated in the loop at lines 8-12. Then the emulator reports the possible advance on its own timeline at lines 14-18.

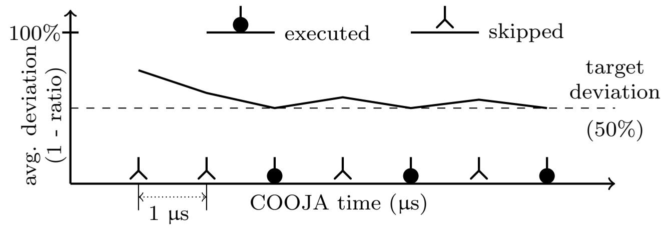
4.2. Execution with deviation

In the simple execution model presented in Section 4.1, the simulation and the motes timelines were constantly synchronized. When clock drift is involved however, the simulator deliberately desynchronizes the mote-specific timeline from the simulation timeline to emulate a deviation of the mote clock from its base frequency. This can be configured in COOJA with the *deviation* parameter which this section describes.

The goal of this parameter is to run each mote on its own timescale, hence simulating a parametrable clock drift per mote. The resulting clock drift would be observable on all clock sources used in the microcontroller. The mote emulator supports two different clock sources, the main one driving the execution of instructions and a real-time clock. Both clocks are equally influenced by the deviation. Using the same notation as Section 2.1, it is the value of y_0 considering the other parameters of the model $x_0 = D = \varepsilon_x(t) = 0$. Its value cannot be greater than one. As a result it is only possible to run motes clocks slower than their base frequency.

The implementation of the deviation is a challenging task. The simulation can run at a microsecond granularity for multiple motes. Hence the complexity of the implementation would greatly impact the running time of the simulation as a whole. Thus, to limit the complexity, the simulation and motes timelines are all discretes. The deviation parameter, however, is a floating number and rounding errors accumulate for each execution step. Also the duration for each execution step is not constant when sleep is enabled. Still the implementation must maintain the microsecond precision for all other motes in the simulation.

The gray parts in Listing 1 present the current mechanism handling the deviation parameter in COOJA.



skipped	1	2	3	3	4	4	5
total	4	5	6	7	8	9	10
ratio	25%	40%	50%	43%	50%	44%	50%

Figure 4: Convergence of the average deviation ratio.

In the implementation the execution speed of the mote is adjusted by skipping the execution until the requested deviation ratio is reached. Two counters, `skipped` and `total`, retain the number of skipped and the total number of execution steps which are the COOJA equivalent of the mote Δ µsevaluation period presented in Figure 3. These values are then used to compute the average deviation ratio at the beginning of each execution and decide whether to skip the current execution step or not.

This mechanism is illustrated on Figure 4. Steps are skipped as long as the two counters verify the inequation

$$\frac{\text{skipped}}{\text{total}} < 1 - \text{deviation} \tag{6}$$

Thus the average deviation ratio, $1 - \frac{\text{skipped}}{\text{total}}$, quickly converges to the target deviation of 50%. Once the target deviation is reached, it is easy to observe that the error to the target deviation stays less or equal to $\frac{1}{total}$.

There are multiple remarks worth noting concerning this implementation. Since the implementation can only increment the number of skipped execution step, it only works for deviation values below the base frequency. Secondly it assumes that the timeline advances with a constant step, in this case Δ µs. This is not the case when low power modes are enabled and the mote can enter into a sleeping state for long durations. This is the origin of the effective clock drift inaccuracies which we explain in the next section.

5. Model inaccuracies

In this section, we present an experiment to assess the validity of the current implementation for the simulation of the clock drift in COOJA. We observe inaccuracies and pinpoint their origin.

5.1. Experimental validation of clock drift

The goal of our experiment is to measure the actual clock drift observed in simulation by adjusting the deviation parameter. To do so, we measure at regular interval the number of CPU cycles elapsed since the mote started and we correlate this value on the simulation timeline in microseconds. We then use linear regression to infer the observed frequency for the simulated mote, thus the observed deviation from its base frequency.

For this experiment, we wrote a custom MSP430 firmware with a recurring parametrable sleep period. Our measurement of the number of CPU cycles happens at each node wake up. We also made sure that the firmware does not change the CPU clock speed after it has been configured. The CPU is configured to a frequency of 3.9 MHz and stays the same for all the simulations in our study.

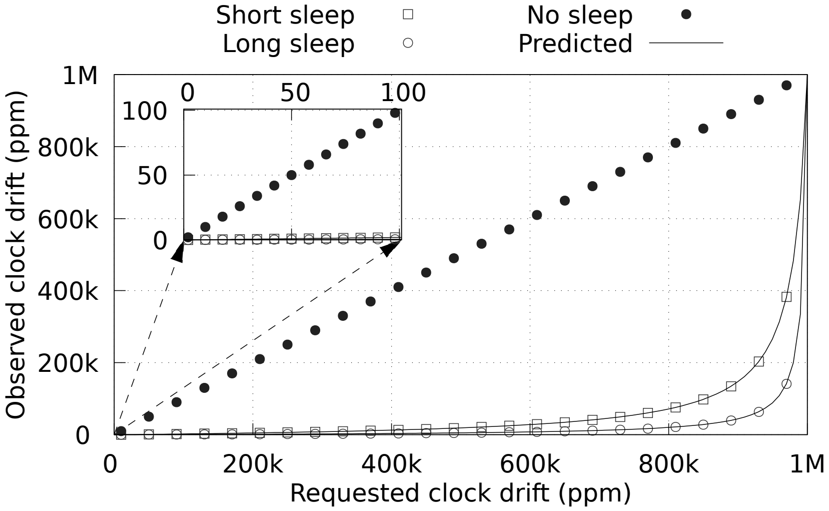


Figure 5: Comparison of observed to requested drift with a zoom on more realistic drift values from 0 to 100 ppm.

We ran this experiment for values of the drift y_0 from 0 to 99%. Recall that in COOJA the deviation d is expressed as $d = 1 + y_0$, hence this translates to deviation parameter values from 100% down to 1%. Each simulation was run for one hour simulation time. We then compared the observed deviation from the base frequency to the requested deviation. Figure 5 shows this comparison, expressed in parts per million (ppm) drift from the base frequency (bottom and left axes).

To reduce energy consumption, a firmware can switch the MCU to a sleep operating mode known as Low Power Mode (LPM). Contiki OS for example does this as soon as there is no event left on its scheduler. Figure 5 shows that when this LPM is disabled the observed clock drift exactly matches the value requested by the configured deviation parameter (see dataset “No sleep”).

However when LPM is enabled, the observed clock drift lies far below its expected value. In our experiments, the firmware wakes up at a regular interval to toggle an LED and keeps the MCU in LPM for the remaining of the time. To assess the impact of sleep duration, we used two different wake-up intervals : 0.98 ms (“Short sleep”) and 3.91 ms (“Long sleep”) which correspond respectively to 32 and 128 ticks of the real-time clock running at 32768 Hz. We observe that with a requested deviation of 50% from the base

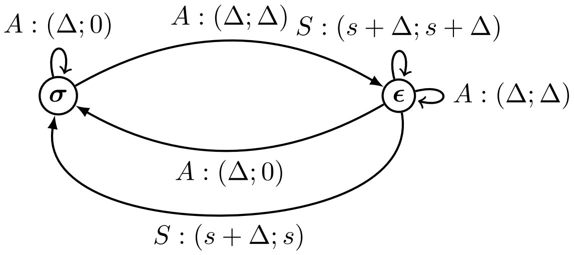


Figure 6: Possible transitions for the simulation steps and their contribution to the simulation and motes timelines.

frequency of 3.9 MHz and with a sleep interval of 0.98 ms, the observed frequency is 3.83 MHz instead of the expected 1.95 MHz. The longer sleep duration of 3.91 ms leads to an even larger error with an observed frequency of 3.88 MHz.

One can argue that deviations of 50% from the base frequency are seldom encountered in real clocks. For this reason, Figure 5 also zooms on smaller drift values ranging from 0 to 100 ppm. For those smaller deviations the error between observed and requested clock drift remains large and grows almost linearly.

5.2. Mathematical model

To further verify our hypothesis on the origin of the clock drift inaccuracies, we build, on the basis of our understanding of the interaction between LPM and the clock drift model, a mathematical model of the simulated clock drift. Our model predicts the magnitude of the clock drift error from the requested deviation d and a variable α that summarizes the LPM behaviour. The plain curves entitled “Predicted” shown on Figure 5 are obtained from our model. They perfectly match the observed values.

The objective pursued here is twofold. First the model must accurately reproduce the observed inaccuracies, giving us a clue that our understanding of the problem is correct. Secondly it allows to estimate the error caused by clock drift inaccuracies in other studies and its impact on their results.

As part of the deviation model presented in Section 4.2, the execution of a mote with deviation can be summarized as a series of skipped steps, denoted σ, and executed steps, denoted ϵ. Skipped and execution steps appear in proportion to the given deviation parameter. For example, the sequence $\sigma \rightarrow \sigma \rightarrow \epsilon \rightarrow \epsilon \rightarrow \epsilon$, where $\rightarrow$ denotes the transition from one simulation step to another, leads to a 60% deviation. The instructions evaluated in each simulation step can trigger a sleep (resp. awake) period during the transition, which we denote by $\xrightarrow{S}$ (resp. $\xrightarrow{A}$).

Figure 6 presents the possible transitions between successive simulation steps. Each edge denotes the time contribution, given in microseconds, to both simulation and mote timelines when going from one simulation step to the next. The label on each edge are as follows. The first letter marks if the transition happens over a sleep period (S) or over an awake period (A). Then follows the contribution in µs to the simulation and mote timeline. For instance, a label $S : (s + \Delta; s)$ means that a sleep period occured. In

Table 1: Transition contribution and probability of occurence.

	active ($\overline{p_s}$)				**sleep** (p_s)	
Transition (i)	$\sigma \xrightarrow{A} \sigma$	$\sigma \xrightarrow{A} \epsilon$	$\epsilon \xrightarrow{A} \epsilon$	$\epsilon \xrightarrow{A} \sigma$	$\epsilon \xrightarrow{S} \epsilon$	$\epsilon \xrightarrow{S} \sigma$
$\boldsymbol{w_i}$	$\bar{d}^2$	$\bar{d}d$	$d^2\overline{p_s}$	$d\bar{d}\overline{p_s}$	d^2p_s	$d\bar{d}p_s$
$\boldsymbol{t_i^s}$	Δ	Δ	Δ	Δ	$s+\Delta$	$s+\Delta$
$\boldsymbol{t_i^m}$	0	Δ	Δ	0	$s+\Delta$	s

the implementation of MspMote $\Delta = 1$ µs. If we assume s to be the duration of the sleep period on this transition, the contribution to the simulation timeline is $s + \Delta$ µs and the contribution to the mote timeline is s µs.

To measure the duration of a sequence of executions, we sum the individual contribution of each transition in the sequence. For instance the sequence $\epsilon \xrightarrow{S} \sigma \xrightarrow{A} \epsilon \xrightarrow{A} \sigma$ in Figure 9 has a simulation time duration of $(s + 3)$ µs and mote time duration of $(s + \Delta)$ µs. It follows easily that the order of the individual transitions in this sequence does not change their combined contribution to the simulation and motes timelines. Thus, any sequence of $m + n$ steps for a simulation with a deviation ratio D is equivalent to the sequence $\underbrace{\sigma \to \sigma \cdots \sigma \to \sigma}_{m} \to \underbrace{\epsilon \to \epsilon \cdots \epsilon \to \epsilon}_{n}$ such that $D = \frac{n}{m+n}$. This assertion stays valid for any combination of transition $\xrightarrow{A}$ or $\xrightarrow{S}$ within the sequence.

We predict the observed deviation ratio D by computing the average contribution to the simulation timeline (T_s) and mote timeline (T_m) for each possible transition in Figure 6. We now consider s to be the average of sleep durations over the entire simulation. Let p_s the probability for a simulation step to result in a sleep period and d the probability that the following step is executed, that is the value of the deviation parameter. We denote $\overline{p_s}$ and $\bar{d}$, the complementary, that is respectively $1 - p_s$ and $1 - d$.

Table 1 lists the individual time contributions in microseconds for each transition i presented in Figure 6 to both the simulation (t_i^s) and mote timelines (t_i^m) along with their probability of occurence (w_i). We calculate this probability by multiplying the individual probability of occurence for each state at the beginning and the end of the transition and the probability to either trigger a sleep period or to stay awake. For instance with $\epsilon \xrightarrow{S} \sigma$, the states involved are an executed step ϵ occurring with a probability d and a skipped step σ occurring with a probability $\bar{d}$. The fact that a sleep period occured on this transition also has a probability p_s. Thus we have $w_{\epsilon \xrightarrow{S} \sigma} = d\bar{d}p_s$.

From this we can compute the average total contribution to the simulation and mote timelines by summing over all the transitions in Figure 6.

$$\begin{aligned} T_m &= \sum_i t_i^m w_i \\ T_s &= \sum_i t_i^s w_i \end{aligned} \tag{7}$$

It follows the predicted observed deviation ratio

$$D(d, s, p_s) = \frac{T_m}{T_s} = \frac{d(sp_s + \Delta)}{dsp_s + \Delta} \tag{8}$$

If we pose $\alpha = sp_s$, a characteristic of the firmware, we can write the predicted deviation:

$$D(d, \alpha) = \frac{d(\alpha + \Delta)}{d\alpha + \Delta} \tag{9}$$

The firmware characteristic must be measured for the firmware under test. For example, the value of α for the predicted value of the short and long sleep in Figure 5 are respectively 50.99 µs and 193.57 µs.

Let $\epsilon_s = \epsilon \xrightarrow{S} \cdots$, the event of a simulation step resulting in a sleep period and ϵ, the event of an executed step as presented in Figure 6. We can write more formally $p_s = P(\epsilon_s)$ where $P(\epsilon_s)$ is the probability of the event ϵ_s and $s = E(S|\epsilon_s)$ where $E(S|\epsilon_s)$ is the mean of the random variable S, the sleep duration, given that ϵ_s occured. It follows a more formal definition of the firmware characteristic α:

$$\alpha = E(S|\epsilon_s)P(\epsilon_s) \tag{10}$$

Frequentially we can compute $E(S|\epsilon_s) = \frac{\sum_i s_i}{\#\epsilon_s}$ and $P(\epsilon_s) = \frac{\#\epsilon_s}{\#\epsilon}$, where $\sum_i s_i$ is the sum of the sleep durations over all simulation steps, and with $\#\epsilon_s$ and $\#\epsilon$ respectively the number of executed steps resulting in a sleep period and the total number of executed steps. It follows:

$$\alpha = \frac{\#\epsilon_s}{\#\epsilon} \frac{\sum_i s_i}{\#\epsilon_s} = \frac{\sum_i s_i}{\#\epsilon} = E(S) \tag{11}$$

Where $E(S)$ is the mean of the sleep period over *all* executed steps (resulting in a sleep period or not). In other words, the α firmware characteristic is the average of the value directly returned by `stepMicros()` in Listing 1. That is, the sleep duration when a sleep occured or 0 otherwise. Thus, the resulting value of α depends both on the real sleep periods programmed in the firmware (which would influence $E(S|\epsilon_s)$), but also on the amount of code executed outside those sleep period (which would influence $P(\epsilon_s)$). The unit of α is the same as the unit used for the time contributions presented in Table 1, and given in microseconds.

5.3. Application to real firmwares

Equation 9 allows to evaluate the clock drift inaccuracies in previous studies without restarting the simulation. However it also requires to compute a firmware specific factor α. We aim to show in the

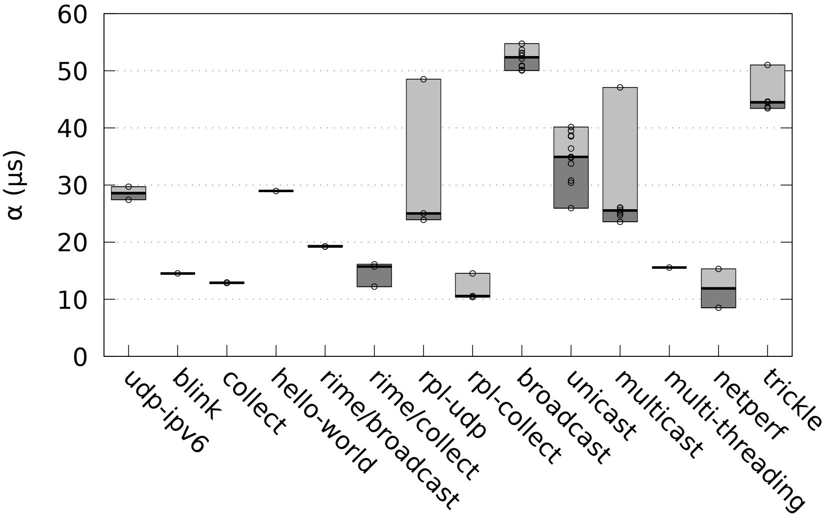


Figure 7: Value of α for selected examples of Contiki firmwares. For each example the median, minimum and maximum value is shown along with the value of α for each mote in the simulation.

remaining paragraphs that the value of α does not vary so much as to significantly change the clock drift error from one firmware to another.

To support this hypothesis we measured the value of α for typical usages of firmwares found in the `examples` directory in the Contiki OS source code. The selected examples range from simple *"Hello World"* applications to more advanced simulations using the various communication components and libraries available in Contiki OS. Figure 7 shows the median, minimum and maximum value of α for the motes in each simulation. In some examples, the sink node may have a different behavior hence resulting in a value of α that is significantly different than the rest of the nodes in the simulation, as can be observed e.g. in firmwares `rpl-udp` and `multicast`.

It remains to be seen how the values presented in Figure 7 affect the clock drift observed in simulation. Figure 8 compares the theoretical observed to requested deviation for the selected firmwares. The full range of α values shown in Figure 7 is included in the gray area of Figure 7. For reference, the dashed line shows the behavior expected for a perfectly simulated clock drift when the observed perfectly matches the requested deviation.

To put these results into perspective, we also look at the relative error on the requested clock drift, expressed in ppm, for the minimum and maximum value of α in all the examples measured. That is, we calculate

$$\varepsilon_{\text{ppm}} = \frac{d_{\text{ppm}} - D_{\text{ppm}}}{d_{\text{ppm}}} \tag{12}$$

where d_{ppm} is the clock drift requested in the simulation and D_{ppm} the observed value for the clock

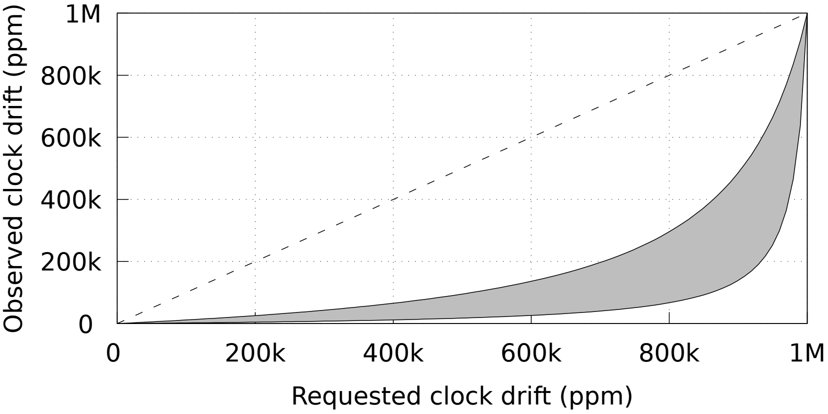
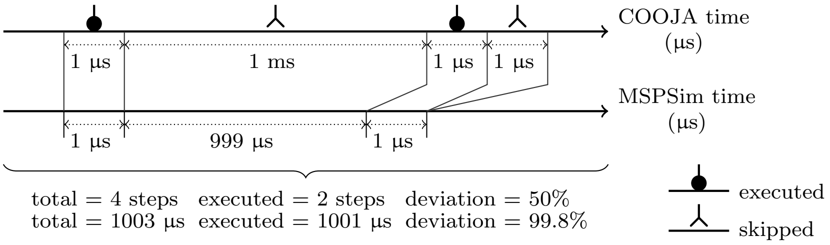


Figure 8: Comparison of observed to requested drift for selected examples of Contiki firmwares. The dashed identity line serves as a reference for the expected behavior of a perfectly simulated clock drift.



COOJA time
(µs)
1 µs
1 ms
1 µs
1 µs
MSPSim time
(µs)
1 µs
999 µs
1 µs
total = 4 steps executed = 2 steps deviation = 50%
total = 1003 µs executed = 1001 µs deviation = 99.8%
executed
skipped

Figure 9: Origin of deviation parameter inaccuracies.

drift.

Table 2 lists the relative error for a requested clock drift of 10 to 30 ppm and a larger clock drift of 1000 ppm. In all cases ε_{ppm} stays above 89%. Thus even without measuring α, one can assume that in most common usages for a Contiki firmware, the resulting clock drift stays far below the requested value with a relative clock drift error of nearly 90% of the requested clock drift.

Table 2: Relative clock drift error (ϵ_{ppm}) for the minimum and maximum value of α in selected Contiki example firmwares.

α (µs)	**10 ppm**	**20 ppm**	**30 ppm**	**1 kppm**
8.52	89.50%	89.50%	89.50%	89.49%
54.76	98.21%	98.21%	98.21%	98.20%

5.4. Inaccuracies origin

From the previous results, we know that the origin of the drift inaccuracies is bound to the sleep mechanism. Further experimentations have shown that the duration of the sleep periods also has an impact on the resulting clock drift.

To understand the origin of these inaccuracies, we take a look at the mote and simulation timelines when a sleep period occurs. This situation is presented in Figure 9 for a target deviation of 50%, although the reasoning also holds for other values of the deviation parameter. For simplicity, we consider that Δ is 1 µs long which is the case of the current implementation.

On this figure, we represent for each step on the COOJA timeline the duration to the last execution. That is the value of `jump` at line 9 in Listing 1. On the MSPSim timeline instead, we represent the value of `jump` passed to `stepMicros()` at line 17. That is after it has eventually been updated by the deviation condition at lines 11-14. If the condition is verified, the current step is considered as *skipped* and the value of `jump` decremented by 1 µs. Otherwise, the step is considered as *executed* and the value of `jump` left unchanged. It is this jump value which, passed to the emulator, dictates the advance of time on the mote timeline.

In the example, one of the step follows a sleep period resulting in a jump value of 1 ms. The other steps are normal execution resulting in jump values of 1 µseach. Counting the number of execution steps we have 2 *executed* over 4 steps in total, thus reaching a deviation of 50% from the point of view of the implementation. However, computing the ratio of time elapsed on both timelines, we have 1001 µsof executed mote time over 1003 µsof total simulation time, thus reaching an actual deviation of 99.8%. Thus, the origin of the inaccuracies lies in the discrepancies between the implementation assumption of execution steps with a constant duration of 1 µs, and sleep periods which are generally much larger than that.

6. Accurate drift model

In this section, we present a new algorithm to implement the deviation parameter. Then we evaluate its performance and show that it can accurately reproduce the requested clock drift for the same time complexity.

The implementation of the deviation parameter in COOJA has the particularity of being independent from the underlying mote emulators. We follow the same approach and implement the new algorithm in the COOJA execution model. Thus, it does not require modification in the motes emulators code, nor extensive changes inside COOJA. Also, since the usage of the deviation parameter does not change, it is possible to reuse the simulation setup of previous studies without any adaptation.

The current implementation uses step counters to compute the average deviation ratio and decide whether to skip or execute the current execution step. When sleep is involved, however, execution steps have a variable duration resulting in an incorrect deviation. Our solution uses a different approach and applies the deviation directly on the mote and simulation timelines carefully handling the resulting rounding errors.

This new algorithm continuously adjusts the advance in the mote timeline to reproduce the requested clock drift. Since the timelines are discrete, rounding errors need to be compensated to avoid their accumulation during the simulation. To do so, rounding errors are carried on the following execution steps.

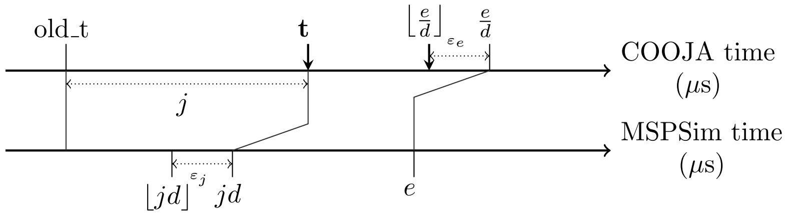


Figure 10: New implementation of the deviation parameter.

Figure 10 illustrates a general working case for our new implementation starting with a mote execution scheduled at time t. At this point the simulator knows the duration since the last execution of the mote. Thus in the absence of clock deviation it must advance the mote timeline for a duration $j = \mathrm{t} - \mathrm{old\_t}$. When deviation exists this jump is corrected by multiplying j with the deviation parameter d and rounded down resulting in a jump error ε_j. It is important that the correction be rounded down instead of up so that the resulting advance in the mote timeline does not fall beyond the emulator internal cycles limit. That is, that the value provided to the emulator does not jump beyond the next internal event or the current cycles count value. Doing so would result in an immediate abort of the simulation.

The executed instructions would eventually program an event e at some point in the future on the mote timeline and switch the CPU off until then. The emulator would return the duration to this event in term of its mote timeline to the simulator. Hence the simulator must anew correct this value taking into account the clock deviation by dividing e with the deviation parameter d, and schedule the next execution for this mote. Again this value must be rounded down to fit the discrete timeline resulting in an execution error ε_e.

With these double corrections alone, the rounding errors ε_j and ε_e would accumulate on each execution, quickly deviating from the requested clock drift. To counteract this effect for ε_j the algorithm keeps track of the accumulated error and carries a single timeline step when necessary. For ε_e the error would result in a mote scheduled earlier in the simulation than intended by the mote. Hence on the next mote execution, the advance on the mote timeline would end before the planned next mote event e. This gap would be reported by the emulator and carried on the next execution as a single step thus bounding the accumulated error ε_e similarly to ε_j.

With these two mechanisms the accumulated error for ε_e on the simulation timeline and ε_j on the mote timeline always stays below 1 µs. The combination results in an error $\varepsilon_S = \varepsilon_e + \frac{\varepsilon_j}{d}$ on the simulation timeline. Thus at any point in the simulation, the instantaneous error for the execution of a particular mote on the simulation timeline is at worst $1 + \frac{1}{d}$ µs.

Listing 3 presents a simplified version of our new drift algorithm integrated in the COOJA execution model. Similarly to the previous implementation, the simulator retains the last mote execution in the global variable `old_t`. Lines 8-10 apply the deviation parameter to the advance on the mote timeline. Line 12 accumulates the jump error ε_j. Once the accumulated jump error exceeds 1 µs, lines 15-16 carry 1 µson the current jump value, thus keeping the accumulated error below 1 µs. Again, line 16 triggers an execution step of Δ µsand provide MSPSim with the advance on its mote timeline. Line 20 converts the value returned by MSPSim from the mote back to the simulation timeline. Finally the simulator uses this value to schedule the next mote execution on line 22.

Again these changes only impact the mote execution model in COOJA and do not require any modification on the emulator side. Hence, they are non-intrusive and remain compatible with other supported emulators.

6.1. Achieved accuracy

We assessed the accuracy of the new implementation by conducting the same experiments presented in Section 5. In the results that follow we report the absolute clock drift error, that is the absolute difference in ppm between the clock drift value requested via the deviation parameter and the clock drift value observed at the end of a one hour simulation.

Figure 11 presents the clock drift error for very large values of the requested clock drift up to 1 million ppm. For all measurements the absolute clock drift error stays below 2.5×10^{-6} ppm. For requested clock drift values below 500000 ppm, the resulting clock drift error is even lower and stays below 1.5×10^{-10} ppm. Above this value, the drift gets so large that all the clocks run very slowly. Hence, not only are the time measurement from the firmware less precise, but only a very small subset of the measurements points are available to estimate the observed clock drift, resulting in a larger error.

```
 1 MOTE_PARAMETER(deviation)
 2
 3 old_t     = 0
 4 jumpError = 0.0
 5 Δ = 1 # µs
 6
 7 def execute(t):
 8   jump      = t - old_t
 9   exactJump = jump * deviation
10   jump      = floor(exactJump)
11
12   jumpError += exactJump - jump
13
14   if jumpError > 1.0:
15     jump += 1
16     jumpError -= 1.0
17
18   step_duration  = MSPSim.stepMicros(jump, Δ)
19   step_duration += Δ
20   step_duration  = floor(step_duration / deviation)
21
22   scheduleNextExecution(t + step_duration)
23   old_t = t
```

Listing 3: New implementation of the deviation parameter.

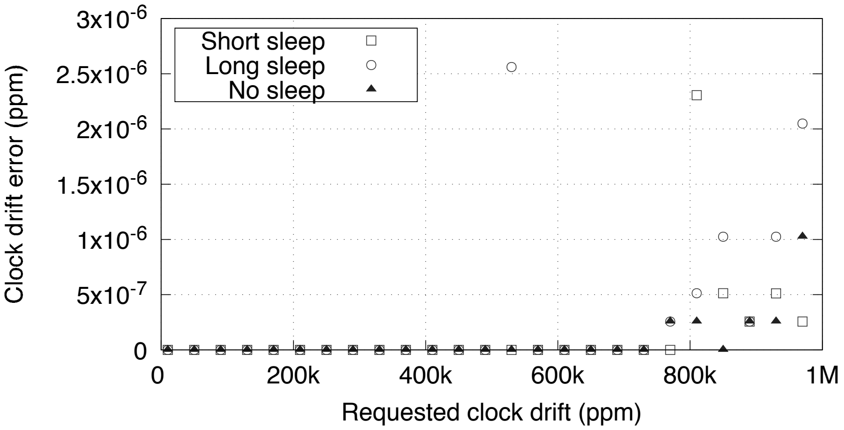


Figure 11: Clock drift error with the new drift algorithm.

For smaller, more realistic values of the clock drift in the range 0 to 100 ppm, the absolute clock drift error is 0 ppm. That is, we measure no error. At this point the absolute clock drift error is so small that it falls below the machine floating point precision.

6.2. Impact on simulation time

We also look at the time complexity of the new implementation. Figure 12 compares the duration of both the new and old implementation for 100 runs of a 600 s simulation and a sleep duration of 0.98 ms. This comparison is done for three values of the deviation parameter, 75 % of the base frequency, 10ppm from the base frequency and no clock drift at all with a deviation parameter of 100 %.

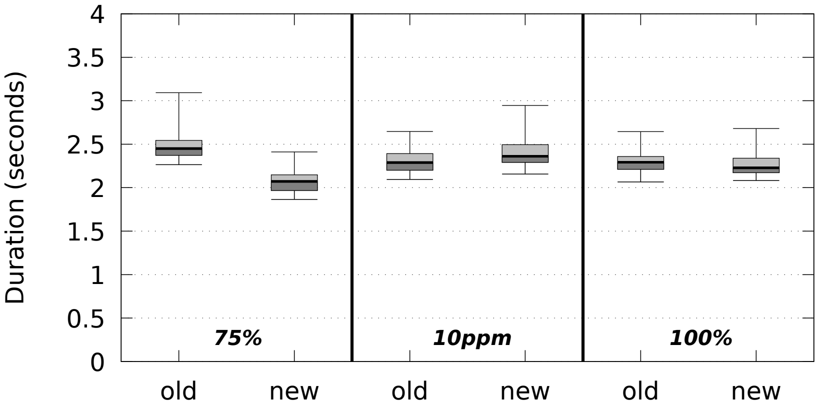


Figure 12: Duration of the execution of the legacy and new clock drift models for various values of the deviation. Minimum, 1$^{\mathrm{st}}$ quartile, median, 3$^{\mathrm{rd}}$ quartile and maximum are represented for each set of runs.

In all cases, the complexity of both algorithms stays in the same order of magnitude. However, with a deviation parameter of 75 % the new implementation is faster than the older version. In the old implementation the observed clock drift for a deviation of 75 % would be much lower than requested. As a result the node clock would run at a much higher frequency than initially intended. Hence, the additional execution steps contribute to the higher complexity of the simulation run.

With a smaller clock drift of 10 ppm instead, the new implementation is slightly slower. In this case, the complexity of the new algorithm outweighs the time benefit of a more accurate clock drift. However, since most simulations do not require clock deviation, the implementation bypasses all computations when no drift is requested. As a result the new implementation is also slightly faster in the common simulation scenario with the deviation parameter to its default value of 100 %, that is, when no clock drift is requested.

7. Application: ContikiMAC Blackouts

In this section, we compare the new and legacy clock drift in a simulation experiment that aims to reproduce a behavior induced by the clock drift in real nodes. In particular we aim to show that the legacy clock drift cannot reproduce the behavior in simulation while our new implementation can.

In 2016, Uwaze et al. discovered PDR and latency inconsistencies for messages exchanged using the ContikiMAC RDC protocol [UBT$^+$16, BSTS17]. These inconsistencies appeared as periodic consecutive loss of packets called *blackouts* and were observed in a real testbed but not in simulation. Careful analysis of the radio traffic pinpointed the origin of these blackouts to recurring timing errors in the operation of ContikiMAC combined with the relative clock drift that happens between real nodes. In this case, the drift causes the phase of the sender and receiver to become orthogonal hence preventing the reception of packets for a certain period.

With ContikiMAC, motes wake up asynchronously following a common duty cycle. When a mote has a frame to send, it transmits it repeatedly until an ACK frame is received and does so at most for the duration of a wake-up cycle. Since the phase difference between sender and receiver is arbitrary, a receiver may wake up at any time during the sender transmission. The receiver detects an incoming frame by performing two consecutive Clear Channel Assessments (CCA). If any of the two CCAs detects a signal, the receiver remains awaken to catch the incoming frame. Then, if the receiver is the destination, it sends an ACK frame.

In order for this scheme to work properly, some timing constraints must be satisfied, as illustrated on Figure 13. The upper part of this figure zooms in on a single packet transmission attempt by ContikiMAC. To ensure that a receiver has the opportunity to transmit an ACK frame, two consecutive DATA frames must be spaced by a time t_i which is longer than the ACK transmission time. The time between the receiver two CCAs is t_c. This time must be chosen so that two consecutive CCAs cannot fall between two DATA frames. The CCA length is t_r. Finally, we denote by t_s the transmission duration of a DATA frame.

The lower part of Figure 13 shows multiple consecutive transmission attempts with a relative clock drift between the sender and receiver. The status of the transmission is marked as ✓ if the transmission was successful or ✗ otherwise. We see that the position of the receiver CCAs drifts along each transmission

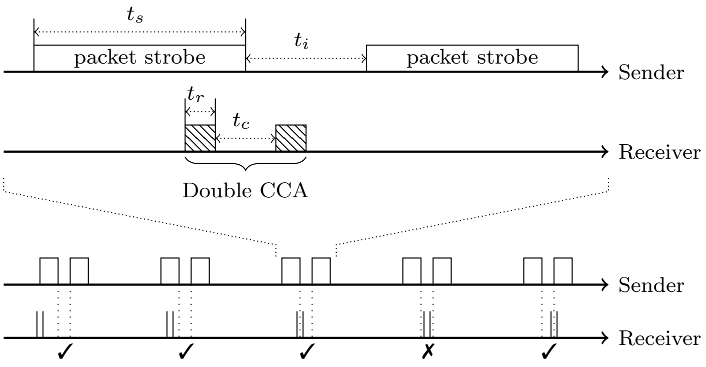


Figure 13: ContikiMAC timing parameters and blackout occurence.

attempt. The clock drift also affects the value of t_s, t_i, t_r and t_c and can break the timing contraints required by ContikiMAC. In particular it allows two CCAs to fit between packet strobes. For the first three and the fifth attempts, one of the receiver CCAs is performed during a sender packet strobe and thus detects the transmission. For the fourth attempt, the two CCAs fall between the sender packet strobes and thus do not detect the transmission. A blackout period ensues for as long as the two CCAs still fit between the packet strobes.

According to Uwaze et al. study of the problem, the blackout would last $(t_i - t_c)/\delta f$ and repeat every $(t_s + t_i)/\delta f$ seconds, where δf is the relative clock drift. We tried to reproduce the same experiment in COOJA. To this end, we implemented a simple IPv6 UDP ping application in Contiki OS that sends each second a fixed packet of 65 Bytes from a sender to a receiver node. To ensure that the traffic never changes, we disabled RPL and IPv6 periodic neighbor discoveries and announcements. We also disabled CSMA to measure the behavior of ContikiMAC itself. Table 3 presents the measured values for t_s, t_i, t_c and t_r from our simulation.

Table 3: Measured value of the ContikiMAC parameters for the blackouts experiment in COOJA.

Parameter	**Value (µs)**
t_s	2082 ± 13
t_i	1367 ± 6
t_c	612 ± 0
t_r	333 ± 1

We use the COOJA deviation parameter to induce different clock drifts on the receiver node. Then we identify recurring loss of packets in the trace and measure their periodicity and duration. Table 4 lists the characteristics of those blackouts for a clock drift parameter of 50 and 20 ppm with the new drift implementation. The simulation also allowed us to better refine the theoretical model from Uwaze et al. For the duration of the blackouts we now have $(t_i - t_c - t_r)/\delta f$.

A special case arises when the drift causes the two CCAs to fall on the last packet strobe of a send attempt. Since the packet strobe is not followed by another one, the receiver cannot receive the packet fully, hence a packet loss occurs. The clock drift causes this situation to repeat periodically adding a second mode to the periodicity of the blackouts. This would cause a significant variance on the measured blackouts period. We filtered those cases in the period values presented in Table 4.

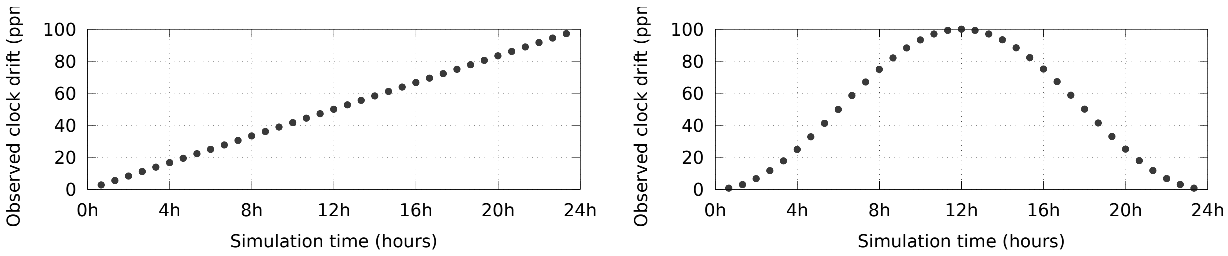
Table 4: Periodicity and duration of the blackouts for the legacy and new implementation of the clock drift.

	New (s)	**Theory (s)**	**Refined (s)**
	50 ppm		
Period	69.18 ± 0.55	68.98	68.98
Duration	6.64 ± 1.05	15.10	8.44
	20 ppm		
Period	172.70 ± 1.31	172.44	172.44
Duration	17.93 ± 2.36	37.76	21.1

While the blackouts characteristics with the new drift implementation match the theoretical value observed on real nodes, the same cannot be said for the legacy drift implementation. Indeed, in the latter case, we did not observe regular loss of packets, hence resulting in a totally different outcome for the experiment.

8. Dynamic drift models

The current implementation of the deviation parameter only allows to configure the deviation once for the entire duration of the simulation. Although this is useful to evaluate the impact of specific values of the clock drift, it does not faithfully reproduce the behavior of a mote when the clock drift evolves over time. In particular for experiments of long duration, the diurnal temperature variation can have a large impact on the clock drift in the mote oscillators.



(a) Linear variation of 100 ppm per 24h.

(b) Periodic variation of 100 ppm per 24h.

Figure 14: Evolution of the observed deviation over time with the dynamic drift models.

Using our new implementation of the deviation parameter, we implement two simple models of the drift evolving over time. These can be selected in COOJA along with the rate and extent of the clock drift variation. We measure the evolution of the observed deviation over a period of 24 hours for the two models.

Figure 14a shows the evolution of the observed deviation for the linear model. The model is configured to increase the clock drift at a rate of 100 ppm per 24h. Figure 14b shows the evolution of the observed deviation for the periodic model. The model is configured to change the clock drift with an amplitude of 100 ppm in 24h. In both cases the pattern continues indefinitely at the same rate.

The linear model can assess the resilience of protocols to increasing amount of clock drift. We use this approach to evaluate ContikiMAC with the same setup presented in Section 7. In this case, we do not focus on the occurence of a recurring period of blackouts with a moderate clock drift. Instead we continuously increase the clock drift at a rate of 100 ppm per second and determine the amount of deviation until the protocol breaks down. The protocol starts to regularly lose packets at a deviation of 39.1% from the base frequency. At a deviation of 46.7% from the base frequency, all packets are systematically lost.

9. Conclusion

Modeling clock drift is important to assess the validity of time-dependent algorithms and protocols such as for example Radio-Duty Cycling MAC protocols. To this end, some network simulators provide a means to make the clocks of nodes deviate from each other. This is the case with COOJA, one of the most frequently used simulator in the field of Wireless Sensor Networks. In this paper, we evaluate experimentally the clock drift model of COOJA. We show that in some cases, the observed clock deviation lies far from the requested one. Our results show errors as high as 90 % even for small requested deviations in the ppm range.

We identify the origin of these inaccuracies in an unsuspected interaction with modeling the low-power mode (LPM) of nodes. LPM is used extensively by most WSN firmwares to save on energy consumption. As a consequence, we suspect that a large fraction of simulations are affected by these inaccuracies. To understand the origin of clock deviation inaccuracies, we explain the inner workings of COOJA and its interaction with the MSPSim emulator. We also discuss the design decisions made by the authors of these tools to make the simulation of a large number of nodes scalable, such as using discrete timelines. We confirm our understanding by building an analytical model of the clock deviation error that we confront to our experimental data and show perfect match. In the end, we show that the clock deviation errors depend on the sleeping behaviour of the firmware, but in any case remain in the order of more than 89 % of the requested drift.

We then propose, implement and validate a modified algorithm to support accurate clock deviations within COOJA. We show that our algorithm keeps the error on the deviation no more than 2.5×10^{-6} ppm for all values of the requested clock drift. We also show that the improved algorithm only slightly affects the real-time duration of simulations. In the future this could also be applied to other simulators such as the newcoming Renode or extended with a more complete drift model. To show the benefit of our new model, we apply it to reproduce a synchronization issue in a well-known RDC protocol, ContikiMAC, that causes periodic blackouts. To date, this issue was observed in a testbed but was previously impossible to reproduce through simulation.

Finally we use our new algorithm to implement dynamic clock drift models to simulate the behavior of clock drift evolving through time. We illustrate this using our linear model to show the behavior of ContikiMAC with increasing clock deviation. As could be expected the protocol shows its limits, albeit at a significantly high amount of clock deviation of 39.1% from the base frequency. As further work, we plan to study the impact of dynamic clock drift models on the simulation of other well known protocols.

Acknowledgments

This work was supported by the European Regional Development Fund (ERDF) through the IDEES project. We thank Aris Koustiamanis, Maximilien Charlier, Georgios Papadopoulos and Jacques Tiberghien for their comments on previous versions of this manuscript.

[A+87] D. W. Allan et al. Time and frequency(time-domain) characterization, estimation, and prediction of precision clocks and oscillators. *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, 34(6):647–654, 1987.

[AHWR18] S. Alharby, N. Harris, A. Weddell, and J. Reeve. Impact of duty cycle protocols on security cost of IoT. In *Information and Communication Systems (ICICS), 2018 9th International Conference on*, pages 25–30. IEEE, 2018.

[B+71] J. A. Barnes et al. Characterization of frequency stability. *IEEE transactions on instrumentation and measurement*, 1001(2):105–120, 1971.

[BSTS17] M. Bezunartea, B. Sartori, J. Tiberghien, and K. Steenhaut. Tackling malfunctions caused by Radio Duty Cycling protocols that do not appear in simulation studies. In *Proceedings of the First ACM FAILSAFE Workshop*, pages 10–15, 2017.

[CS66] Leonard S Cutler and Campbell L Searle. Some aspects of the theory and measurement of frequency fluctuations in frequency standards. *Proceedings of the IEEE*, 54(2):136–154, 1966.

[Dun11] A. Dunkels. The ContikiMAC Radio Duty Cycling Protocol. Technical report, SICS, 2011.

[EDF+16] A. Elsts, S. Duquennoy, X. Fafoutis, G. Oikonomou, R. Piechocki, and I. Craddock. Microsecond-accuracy time synchronization using the IEEE 802.15.4 TSCH protocol. In *Proceedings of IEEE SenseApp*, 2016.

[FCF07] A. Fraboulet, G. Chelius, and E. Fleury. Worldsens: development and prototyping tools for application specific wireless sensors networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, 2007.

[FMT10] F. Ferrari, A. Meier, and L. Thiele. Accurate clock models for simulating wireless sensor networks. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, pages 21:1–21:4, 2010.

[GHB+19] David Griffin, James Harbin, Alan Burns, Iain Bate, Robert I Davis, and Leandro Soares Indrusiak. Validating high level simulation results against experimental data and low level simulation: a case study. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, pages 30–40, 2019.

[HQ19] David Hauweele and Bruno Quoitin. Toward accurate clock drift modeling in wireless sensor networks simulation. In *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 95–102, 2019.

[iee15] IEEE Standard for Local and metropolitan area networks - Part 15.4. *IEEE Std. 802.15.4-2015*, 2015.

[IVML18] P. Ioulianou, V. Vasilakis, I. Moscholios, and M. Logothetis. A signature-based intrusion detection system for the internet of things. *ICTF 2018*, 2018.

[KKCP17] H.-s. Kim, J. Ko, D. Culler, and J. Paek. Challenging the ipv6 routing protocol for low-power and lossy networks (rpl): A survey. *IEEE Communications Surveys and Tutorials*, 19:2502–2525, 09 2017.

[KPK+18] A. Kokkinis, A. Paphitis, L. Kanaris, C. Sergiou, and S. Stavrou. Physical and network layer interconnection module for realistic planning of IoT sensor networks. In *EWSN*, 2018.

[Ltd18] Antmicro Ltd. Renode - documentation. `https://media.readthedocs.org/pdf/renode/latest/renode.pdf`, Sept 2018.

[MDV+18] Esteban Municio, Glenn Daneels, Mališa Vučinić, Steven Latré, Jeroen Famaey, Yasuyuki Tanaka, Keoma Brun, Kazushi Muraoka, Xavier Vilajosana, and Thomas Watteyne. Simulating 6TiSCH networks. *Transactions on Emerging Telecommunications Technologies*, 2018.

[MPRS16] I. Minakov, R. Passerone, A. Rizzardi, and S. Sicari. A comparative study of recent wireless sensor network simulators. *ACM Transactions on Sensor Networks*, 12(3):20:1–20:39, July 2016.

[ODE+06] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and Th. Voigt. Cross-level sensor network simulation with COOJA. In *Proceedings of the 31st IEEE conference on Local computer networks*, 2006.

[PMF+16] G. Z. Papadopoulos, A. Mavromatis, X. Fafoutis, N. Montavont, R. Piechocki, Th. Tryfonas, and G. Oikonomou. Guard time optimisation and adaptation for energy efficient multi-hop TSCH networks. In *Proceedings of IEEE WF-IoT*, 2016.

[PPS+18] A. P. Plageras, K. E. Psannis, Ch. Stergiou, H. Wang, and B. B. Gupta. Efficient IoT-based sensor BIG data collection–processing and analysis in smart buildings. *Future Generation Computer Systems*, 82:349–357, 2018.

[RPGJ18] N. Ramachandran, V. Perumal, S. Gopinath, and M. Jothi. Sensor search using clustering technique in a massive iot environment. In *Industry Interactive Innovations in Science, Engineering and Technology*, pages 271–281. Springer, 2018.

[UBT+16] M.-P. Uwase, M. Bezunartea, J. Tiberghien, J.-M. Dricot, and K. Steenhaut. Poster: ContikiMAC, some critical issues with the CC2420 radio. In *EWSN*, pages 257–258, 2016.

[WVK+12] Th. Watteyne, X. Vilajosana, B. Kerkez, F. Chraim, K. Weekly, Q. Wang, S. Glaser, and K. Pister. Openwsn: A standards-based low-power wireless development environment. *ETT*, 23:480–493, 08 2012.

[YH04] W. Ye and J. Heidemann. Medium access control in wireless sensor networks. In *Wireless sensor networks*, pages 73–91. Springer, 2004.